The KB paradigm and its application to interactive configuration.

Pieter Van Hertum, Ingmar Dasseville,
Gerda Janssens and Marc Denecker

Dept. Computer Science KU Leuven,
{*first*}.{*lastname*}@cs.kuleuven.be

Abstract. The knowledge base paradigm aims to express domain knowledge in a rich formal language, and to use this domain knowledge as a knowledge base to solve various problems and tasks that arise in the domain by applying multiple forms of inference. As such, the paradigm applies a strict separation of concerns between information and problem solving. In this paper, we analyze the principles and feasibility of the knowledge base paradigm in the context of an important class of applications: interactive configuration problems. In interactive configuration problems, a configuration of interrelated objects under constraints is searched, where the system assists the user in reaching an intended configuration. It is widely recognized in industry that good software solutions for these problems are very difficult to develop. We investigate such problems from the perspective of the KB paradigm. We show that multiple functionalities in this domain can be achieved by applying different forms of logical inferences on a formal specification of the configuration domain. We report on a proof of concept of this approach in a real-life application with a banking company.

1 Introduction

In this paper, we investigate the application of knowledge representation and reasoning to the problem of *interactive configuration*. In the past decades enormous progress in many different areas of computational logic was obtained. This resulted in a complex landscape with many declarative paradigms, languages and communities. One issue that fragments computational logic is the reasoning/inference task. Computational logic is divided in different declarative paradigms, each with its own syntactical style, terminology and conceptuology, and designated form of inference (e.g, deductive logic, logic programming, abductive logic programming, databases (query inference), answer set programming (answer set generation), constraint programming, etc.). Yet, in all of them declarative propositions need to be expressed. Take, e.g., "each lecture takes place at some time slot". This proposition could be an expression to be deduced from a formal specification if the task was a verification problem, or to be queried in a database, or it could be a constraint for a scheduling problem. It is, in the first place, just a piece of information and we see no reason why depending on the task to be

solved, it should be expressed in a different formalism (classical logic, SQL, ASP, MiniZinc, etc.).

The Knowledge Base (KB) paradigm [8] was proposed as an answer to this. The KB paradigm applies a strict separation of concerns to information and problem solving. A KB system allows to store information in a knowledge base, and provides a range of inference methods. With these inference methods various types of problems and tasks can be solved using the *same knowledge base*. As such the knowledge base is neither a program nor a description of a problem, it cannot be executed or run. It is nothing but information. However, this information can be used to solve multiple sorts of problems. Stated differently, many declarative problem solving paradigms are mono-inferential: are based on one form of inference. Instead, the KB-paradigm is multi-inferential. We believe that this implements a more natural, pure view of what declarative logic is aimed to be. The FO($\cdot$) KB project [8] is a concrete project that runs now for a number of years. Its aim is to integrate different useful language constructs and forms of inference from different declarative paradigms in one rich declarative language and a KB system. So far, it has led to the KB language FO($\cdot$) and the KB system IDP which were used in the configuration experiment described in this paper.

An interactive configuration (IC) problem [9] is an interactive version of a constraint solving problem. One or more users search for a configuration of objects and relations between them that satisfies a set of constraints. Industry abounds with interactive configuration problems: configuring composite physical systems such as cars and computers, insurances, loans, schedules involving human interaction, webshops (where clients choose composite objects), etc. However, building such software is renown in industry as difficult and no broadly accepted solution methods are available [3]. Building software support using standard imperative programming is often a nightmare, due to the fact that (1) many functionalities need to be provided, (2) they are complex to implement, and (3) constraints on the configuration tend to spread out over the application, in the form of snippets of code performing some computation relative to the constraint (e.g., context dependent checks or propagations) often leading to an unacceptable maintenance cost. This makes interactive configuration an excellent domain to illustrate the advantages of declarative methods over standard imperative or object-oriented programming.

Our research question is: can we express the constraints of correct configurations in a declarative logic and provide the required functionalities by applying inference on this domain knowledge? This is a KRR question albeit a difficult one. In the first place, some of the domain knowledge may be complex. For an example in the context of a computer configuration problem, take the following constraint: *the total memory usage of different software processes that needs to be in main memory simultaneously, may not exceed the available RAM memory.* It takes an expressive knowledge representation language to (compactly and naturally) express such a constraint. Many interactive configuration problems include complex constraints: various sorts of quantification, aggregates (as illus-

trated above), definitions (sometimes inductive), etc. Moreover, an interactive configuration system needs to provide many functionalities: checking the validity of a fully specified configuration, correct and safe reasoning on a partially specified configuration (this involves reasoning on incomplete knowledge, sometimes with infinite or unknown domains), computing impossible values or forced values for attributes, generating sensible questions to the user, providing explanation why certain values are impossible, backtracking if the user regrets some choices, supporting the user by filling in his don't-cares potentially taking into account a cost function, etc.

That declarative methods are particularly suitable for solving this type of problem has been acknowledged before, and several systems and languages have been developed [9,17,21,23].

The first contribution of our work is the analysis of IC problems from a Knowledge Representation point of view. We show that multiple functionalities in this domain can be achieved by applying different forms of logical inferences on a formal specification of the configuration domain. We focus on a study of the different forms of inference, determining the forms of inference in terms of which the different functionalities can be supplied. The second contribution is vice versa: a study of the feasibility and usefulness of the KB paradigm in this important class of applications. The logic used in this experiment is the logic FO($\cdot$) [7], an extension of first-order logic (FO), and the system is the IDP system [6]. We discuss the complexity of (the decision problems of) the inference problems and why they are solvable, despite the high expressivity of the language and the complexity of inference. This research has its origin in an experimental IC system we developed in collaboration with industry.

2 The FO(.) KB project

The language. FO($\cdot$) refers to the class of extensions of first order logic (FO) as is common in logic, e.g. FO(LFP) stands for the extension of FO with a least fixpoint construction [11]. Currently, the language of the IDP system in the project is FO(T, ID, Agg, arit, PF) [7,14]: FO extended with types, definitions, aggregates, arithmetic and partial functions. In this project we will use the subset language FO(T, Agg, arit, PF). Abusing notation, we will use FO($\cdot$) as an abbreviation for this language. Below, we introduce the aspects of the logic and its syntax on which this paper relies.

A *vocabulary* is a set Σ of type, predicate and function symbols. Variables x, y, atoms A, FO-formulas φ are defined as usual. Aggregate terms are of the form $Agg(E)$, with Agg an aggregate function symbol and E an expression $\{(\overline{x}, F(\overline{x}))|\varphi(\overline{x})\}$, where φ can be any FO-formula, F a function symbol and $\overline{x}$ a tuple of variables. Examples are the cardinality, sum, product, maximum and minimum aggregate functions. For example $sum\{(x, F(x))|\varphi(x)\}$ is read as $\Sigma_{x\in\{y|\varphi(y)\}}F(x)$. A *term* in FO($\cdot$) can be an aggregate term or a term as defined in FO. A *theory* is a set of FO($\cdot$) formulas.

A partial set on domain D is a function from D to $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. A partial set is two-valued (or total) if $\mathbf{u}$ does not belong to its range. A *(partial) structure* $\mathcal{S}$ consists of a domain D_τ for all types τ in the vocabulary Σ and an assignment of a partial set $\sigma^{\mathcal{S}}$ to each symbol $\sigma \in \Sigma$, called the interpretation of σ in $\mathcal{S}$. The interpretation $P^{\mathcal{S}}$ of a predicate symbol P with type $[\tau_1, \ldots, \tau_n]$ in $\mathcal{S}$ is a partial set on domain $D_{\tau_1} \times \ldots \times D_{\tau_n}$. For a function F with type $[\tau_1, \ldots, \tau_n] \to \tau_{n+1}$, the interpretation $F^{\mathcal{S}}$ of F in $\mathcal{S}$ is a partial set on domain $D_{\tau_1} \times \ldots \times D_{\tau_n} \times D_{\tau_{n+1}}$. In case the interpretation of σ in $\mathcal{S}$ is a two-valued set, we abuse notation and use $\sigma^{\mathcal{S}}$ as shorthand for $\{\overline{d} | \sigma^{\mathcal{S}}(\overline{d}) = \mathbf{t}\}$. The precision-order on the truth values is given by $\mathbf{u} <_p \mathbf{f}$ and $\mathbf{u} <_p \mathbf{t}$. It can be extended pointwise to partial sets and partial structures, denoted $\mathcal{S} \leq_p \mathcal{S}'$. Notice that total structures are the maximally precise ones. We say that $\mathcal{S}'$ extends $\mathcal{S}$ if $\mathcal{S} \leq_p \mathcal{S}'$.

A total structure S is called *functionally consistent* if for each function F with type $[\tau_1, \ldots, \tau_n] \to \tau_{n+1}$, the interpretation F^S is the graph of a function $D_{\tau_1} \times \ldots \times D_{\tau_n} \mapsto D_{\tau_{n+1}}$. A partial structure $\mathcal{S}$ is functionally consistent if it has a functionally consistent two-valued extension. Unless stated otherwise, we will assume for the rest of this paper that all (partial) structures are functionally consistent.

A domain atom (domain term) is a tuple of a predicate symbol P (a function symbol F) and a tuple of domain elements $(d_1, \ldots, d_n)$. We will denote it as $P(d_1, \ldots, d_n)$ (respectively $F(d_1, \ldots, d_n)$). We say a domain term t of type τ is uninterpreted in $\mathcal{S}$ if $\{d | d \in D_\tau \wedge (t = d)^{\mathcal{S}} = \mathbf{u}\}$ is non-empty.

To define the satisfaction relation on theories, we extend the interpretation of symbols to arbitrary terms and formulas using the Kleene truth assignments [12]. For a theory T and a partial structure $\mathcal{S}$, we say that $\mathcal{S}$ is a model of T (or in symbols $\mathcal{S} \models T$) if $T^{\mathcal{S}} = \mathbf{t}$ and $\mathcal{S}$ is two-valued.

Inference tasks. In the KB paradigm, a specification is a bag of information. This information can be used for solving various problems by applying a suitable form of inference on it.

FO is standardly associated with deduction inference: a deductive inference task takes as input a pair of theory T and sentence φ, and returns $\mathbf{t}$ if $T \models \varphi$ and $\mathbf{f}$ otherwise. This is well-known to be undecidable for FO, and by extension for FO($\cdot$). However, to provide the required functionality of an interactive configuration system we can use simpler forms of inference. Indeed, in many such domains a fixed finite domain is associated with each unknown configuration parameter.

A natural format in logic to describe these finite domains is by a partial structure with a finite domain. Also other data that are often available in such problems can be represented in that structure. As such various inference tasks are solvable by finite domain reasoning and become decidable. Below, we introduce base forms of inference and recall their complexity when using finite domain reasoning. We assume a fixed vocabulary Σ and theory T.

Modelcheck(T, S): input: a total structure S and theory T over the vocabulary interpreted by S; output is the boolean value $S \models T$. Complexity is in **P**.

Modelexpand($T, \mathcal{S}$): input: theory T and partial structure $\mathcal{S}$; output: a model I of T such that $\mathcal{S} \leq_p I$ or *UNSAT* if there is no such I. Modelexpand [24] is a generalization for FO($\cdot$) theories of the modelexpansion task as defined in Mitchell et al. [13]. Complexity of deciding the existence of a modelexpansion is in **NP**.

Optimize($T, \mathcal{S}, t$): input: a theory T, a partial structure $\mathcal{S}$ and a term t of numerical type; output: a model $I \geq_p \mathcal{S}$ of T such that the value t^I of t is minimal. This is an extension to the modelexpand inference. The complexity of deciding that a certain t^I is minimal, is in $\mathbf{\Delta_2^P}$.

Propagate($T, \mathcal{S}$): input: theory T and partial structure $\mathcal{S}$; output: the most precise partial structure $\mathcal{S}_r$ such that for every model $I \geq_p \mathcal{S}$ of T it is true that $I \geq_p \mathcal{S}_r$. The complexity of deciding that a partial structure $\mathcal{S}'$ is $\mathcal{S}_r$ is in $\mathbf{\Delta_2^P}$. Note that we assume that all partial structures are functionally consistent, which implies that we also propagate functional constraints.

Query($\mathcal{S}, E$): input: a (partial) structure $\mathcal{S}$ and a set expression $E = \{\overline{x} \mid \varphi(\overline{x})\}$; output: the set $A_Q = \{\overline{x} \mid \varphi(\overline{x})^{\mathcal{S}} = \mathbf{t}\}$. Complexity of deciding that a set A is A_Q is in **P**.

Approximative versions exist for some of these inferences, with lower complexity [23]. More inferences exist, such as simulation of temporal theories in FO($\cdot$) [4], which were not used in the experiment.

3 Interactive Configuration

In an IC problem, one or more users search for a configuration of objects and relations between them that satisfies a set of constraints.

Typically, the user is not aware of all constraints. There may be too many of them to keep track of. Even if the human user can oversee all constraints that he needs to satisfy, he is not a perfect reasoner and cannot comprehend all consequences of his choices. This in its own right makes such problems hard to solve. The problems get worse if the user does not know about the relevant objects and relations or the constraints on them, or if the class of involved objects and relations is large, if the constraints get more complex and more "irregular" (e.g., exceptions), if more users are involved, etc. On top of that, the underlying constraints in such problems tend to evolve quickly. All these complexities occur frequently, making the problem complex for a human user. In such cases, computer assistance is needed: the human user chooses and the system assists by guiding him through the search space.

For a given IC problem, an IC system has information on that problem. There are a number of stringent rules to which a configuration should conform, and besides this there is a set of parameters. Parameters are the open fields in the configuration that need to be filled in by the user or decided by the system.

3.1 Running example: Domain knowledge

A simplified version of the application in Section 5.1 is used in section 4 as running example. We introduce the domain knowledge of this example here.

Example 1. Software on a computer has to be configured for different employees. Available software includes operating systems, editors and text processors. Each software has a price. Some software is required for other software. If more than one OS is needed, a DualBoot System is required. The software, the requirements, the budgets of the employees and the prices of software can be seen in Table 1.

PriceOf		**PreReq**		**MaxCost**		**IsOS**
software	*int*	*software*	*software*	*employee*	*int*	*software*
Windows	60	Office	Windows	Secretary	100	Windows
Linux	20	LaTeX	Linux	Manager	150	Linux
LaTeX	10					
Office	30					
DualBoot	40					

Table 1. Example data

3.2 Subtasks of an interactive configuration system

Any system assisting a user in interactive configuration must be able to perform a set of subtasks. We look at important subtasks that an interactive configuration system should support.

Subtask 1: Acquiring information from the user The first task of an IC system is acquiring information from the user. The system needs to get a value for a number of parameters of the configuration from the user. There are several options: the system can ask questions to the user, it can make the user fill in a form containing open text fields, dropdown-menus, checkboxes, etc. Desirable aspects would be to give the user the possibility to choose the order in which he gives values for parameters and to omit filling in certain parameters (because he does not know or does not care). For example, in the running example a user might need a LaTeX-package, but he does not care about which OS he uses. In that case the system will decide in his place that a Linux system is required. Since a user is not fully aware of all constraints, it is possible that he inputs conflicting information. This needs to be handled or avoided.

Subtask 2: Generating consistent values for a parameter After a parameter is selected (by the user or the system) for which a value is needed, the system can assist the user in choosing these values. A possibility is that the system presents the user with a list of all possible values, given the values for other parameters and the constraints of the configuration problem. Limiting the user with this list makes that the user is unable to input inconsistent information.

Subtask 3: Propagation of information Assisting the user in choosing values for the parameters, a system can use the constraints to propagate the information that the user has communicated. This can be used in several

ways. A system can communicate propagations through a GUI, for example by coloring certain fields red or graying out certain checkboxes. Another way is to give a user the possibility to explicitly ask "what if"-questions to the system. In Example 1, a user can ask the system what the consequences are if he was a secretary choosing an Office installation. The system answers that in this case a Windows installation is required, which results in a Linux installation becoming impossible (due to budget constraints) and as a consequence it also derives the impossibility of installing LaTeX.

Subtask 4: Checking the consistency for a value When it is not possible/desirable to provide a list of possible values, the system checks that the value the user has provided is consistent with the known data and the constraints.

Subtask 5: Checking a configuration If a user makes manual changes to a configuration, the system provides him with the ability to check if his updated version of the configuration still conforms to all constraints.

Subtask 6: Autocompletion If a user has finished communicating all his preferences, the system autocompletes the partial configuration to a full configuration. This can be done arbitrarily (a value for each parameter such that the constraints are satisfied) or the user can have some other parameters like a total cost, that have to be **optimized**.

Subtask 7: Explanation If a supplied value for a parameter is not consistent with other parameters, the system can explain this inconsistency to the user. This can be done by showing minimal sets of parameters with their values that are inconsistent, or by showing (visualizations of) constraints that are violated. It can also explain to the user why certain automatic choices are made, or why certain choices are impossible.

Subtask 8: Backtracking It is not unthinkable that a user makes a mistake, or changes his mind after seeing consequences of choices he made. Backtracking is an important subtask for a configuration system. Backtracking can be supported in numerous ways. The simplest way is a simple back button, where the last choice a user made is reverted. A more involved option is a system where a user can select any parameter and erase his value for that parameter. The user can then decide this parameter at a later timepoint. Even more complex is a system where a user can supply a value for a parameter and if it is not consistent with other parameters the system shows him which parameters are in conflict and proposes other values for these parameters such that consistency can be maintained.

4 Interactive Configuration in the KB paradigm

To analyze the IC problem from the KB point of view, we aim at formalizing the subtasks of Section 3 as inferences. In this paper we do not deal with user interface aspects. For a given application, our knowledge base consists of a vocabulary Σ, a theory T expressing the configuration constraints and a partial structure $\mathcal{S}$. Initially, $\mathcal{S}_0$ is the Σ-partial structure that contains the domains of

the types and the input data. During IC, $\mathcal{S}_0$ will become more and more precise partial structures $\mathcal{S}_i$ due to choices made by the user. For IC, the KB also contains $L_{\mathcal{S}_0}$, the set of all uninterpreted domain atoms/terms[1] in $\mathcal{S}_0$. These domain terms are the logical formalization of the parameters of the IC problem. Σ and T are fixed. As will be shown in this section, all subtasks can be formalized by (a combination of) inferences on this knowledge base consisting of $\Sigma, T, \mathcal{S}_0, L_{\mathcal{S}_0}$ and information gathered from the user.

Example 2. Continuing Example 1, use vocabulary Σ consisting of types: *software*, *employee* and *int* (integers), predicates *Install(software), IsOS(software)* and *PreReq(software,software)*, functions *PriceOf(software):int, MaxCost (employee):int* and two constants *Requester: employee* and *Cost: int*. The initial partial structure $\mathcal{S}_0$ consists of {*employee*→ {*Secretary, Manager*}, *software* → {*Windows, Linux, LaTeX, Office, DualBoot*}} and interpretations for *MaxCost (employee):int, IsOs(software), PriceOf(software):int* and *PreReq(software, software)* as can be seen in Table 1. The set of parameters $L_{\mathcal{S}_0}$ is {*Requester, Install(Windows), Install(Linux), Install(Office), Install(LaTeX), Install (DualBoot), Cost*}. The theory T consists of the following constraints:

$\forall s1\, s2 :\ Install(s1) \wedge PreReq(s1, s2) \Rightarrow Install(s2).$
// The total cost is the sum of the prices of all installed software.
$Cost = sum\{(s, PriceOf(s))|Install(s)\}.$
$Cost < MaxCost(Requester).$
$\exists s :\ Install(s) \wedge IsOS(s).$
$Install(Windows) \wedge Install(Linux) \Rightarrow Install(DualBoot).$

Subtask 1: Acquiring information from the user Key in IC is collecting information from the user on the parameters. During the run of the system, the set of parameters that are still open, changes. In our KB system, a combination of the inferences introduced in Section 2, which is called a derived inference, is used to calculate this set of parameters.

Definition 1. ***Calculating uninterpreted terms.***
GetOpenTerms($T, \mathcal{S}$***)*** *is the derived inference with input a theory T, a partial structure $\mathcal{S} \geq_p \mathcal{S}_0$ and the set $L_{\mathcal{S}_0}$ of terms. Output is a set of terms such that for every term t in that set, there exist models I_1 and I_2 of T that expand $\mathcal{S}$ for which $t^{I_1} \neq t^{I_2}$. Or formally:*

$$\{l|l \in L_{\mathcal{S}_0} \wedge \{d|(l = d)^{\mathcal{S}'} = \mathbf{u}\} \neq \emptyset \wedge \mathcal{S}' = Propagate(T, \mathcal{S})\}$$

The complexity of deciding whether a given set of terms A is the set of uninterpreted terms is in $\mathbf{\Delta_2^P}$.
An IC system can use this set of terms in a number of ways. It can use a metric to select a specific term, which it can pose as a direct question to the user. It can also present a whole list of these terms at once and let the user pick one to supply a value for. In Section 5.1, we discuss two different approaches we implemented for this project.

[1] In the rest of this paper, a domain atom is treated as a term that evaluates to true or false.

Example 3. In Example 2, the parameters and domains are already given. Assume that the user has chosen the value *Manager* for *Requester*, true for *Install(Windows)* and false for *Install(Linux)*. The system will return $GetOpenTerms(T, \mathcal{S}) = \{$*Install(Office), Install(DualBoot), Cost*$\}$.

Subtask 2: Generating consistent values for a parameter A domain element d is a possible value for term t if there is a model $I \geq_p \mathcal{S}$ such that $(t = d)^I = \mathbf{t}$

Definition 2. ***Calculating consistent values.***
GetConsistentValues*$(T, \mathcal{S}, t)$ is the derived inference with as input a theory T, a partial structure $\mathcal{S}$ and a term $t \in GetOpenTerms(T, \mathcal{S})$. Output is the set*

$$\{t^I |\ I \text{ is a model of } T \text{ expanding } \mathcal{S}\}$$

The complexity of deciding that a set P is the set of consistent values for t is in $\mathbf{\Delta_2^P}$.

Example 4. The possible values in the initial partial structure $\mathcal{S}_0$ are: $\{$*Secretary, Manager*$\}$ for *Requester*, the integers for *Cost* and $\{$*true, false*$\}$ for the others.

Subtask 3: Propagation of information It is informative for the user that he can see the consequences of assigning a particular value to a parameter.

Definition 3. ***Calculating Consequences.***
PosConsequences*$(T, \mathcal{S}, t, a)$ and* ***NegConsequences****$(T, \mathcal{S}, t, a)$ are derived inferences with input a theory T, a partial structure $\mathcal{S}$, an uninterpreted term $t \in GetOpenTerms(T, \mathcal{S})$ and a domain element $a \in GetConsistentValues(T, \mathcal{S}, t)$. As output it has a set C^+, respectively C^- of tuples (q, b) of uninterpreted terms and domain elements. $(q, b) \in C^+$, respectively C^- means that the choice a for t entails that q will be forced, respectively prohibited to be b. Formally,*

$$\begin{aligned} C^+ = \{(q,b) \mid\ & (q = b)^{\mathcal{S}'} = \mathbf{t} \wedge (q = b)^{\mathcal{S}} = \mathbf{u} \\ & \wedge\ \mathcal{S}' = Propagate(T, \mathcal{S} \cup \{t = a\}) \\ & \wedge\ q \in GetOpenTerms(T, \mathcal{S}) \setminus \{t\}\ \} \\ C^- = \{(q,c) \mid\ & (q = c)^{\mathcal{S}'} = \mathbf{f} \wedge (q = c)^{\mathcal{S}} = \mathbf{u} \\ & \wedge\ \mathcal{S}' = Propagate(T, \mathcal{S} \cup \{t = a\}) \\ & \wedge\ q \in GetOpenTerms(T, \mathcal{S}) \setminus \{t\}\ \} \end{aligned}$$

The complexity of deciding whether a set P is C^+ or C^- is in $\mathbf{\Delta_2^P}$.

Example 5. Say the user has chosen *Requester* : *Secretary* and wants to know the consequences of making $Install(Windows)$ true. The output in this case contains $(Install(LaTeX), \mathbf{f})$ in $PosConsequences(T, \mathcal{S}, t, a)$ and $(Install(LaTeX), \mathbf{t})$ in $NegConsequences(T, \mathcal{S}, t, a)$ since this combination

is too expensive for a secretary. Note that there is not always such a correspondence between the positive and negative consequences. For example, when deriving a negative consequence for $Cost$, this does not necessarily imply a positive consequence.

Subtask 4: Checking the consistency for a value A value d for a term t is consistent if there exists a model of T in which $t = d$ that extends the partial structure representing the current state.

Definition 4. ***Consistency Checking.***
CheckConsistency*$(T, \mathcal{S}, t, d)$ is the derived inference with as input a theory T, a partial structure $\mathcal{S}$, an uninterpreted term t and an domain element d. Output is a boolean b that represents if $\mathcal{S}$ extended with $t = d$ still satisfies T. Complexity of deciding if a value d is consistent for a term t is in* ***NP****.*

Subtask 5: Checking a configuration Once the user has constructed a 2-valued structure S and makes manual changes to it, he may need to check if all constraints are still satisfied. A theory T is checked on a total structure S by calling $Modelcheck(T, S)$, with complexity in **P**.

Subtask 6: Autocompletion If a user is ready communicating his preferences (Subtask 1) and there are undecided terms left which he does not know or care about, the user may want to get a full configuration (i.e. a total structure). This is computed by modelexpand. In particular:

$$I = Modelexpand(T, \mathcal{S})$$

In many of those situations the user wants to have a total structure with a minimal cost (given some term representing the cost t). This is computed by optimize:

$$I = Optimize(T, \mathcal{S}, t)$$

Example 6. Assume the user is a secretary and all he knows is that he needs Office. He chooses $Secretary$ for $Requester$ and true for $Install(Office)$ and calls autocompletion. A possible output is a structure S where for the remaining parameters, a choice is made that satisfies all constraints, e.g., $Install(Windows)^S = \mathbf{t}$, $Install(DualBoot)^S = \mathbf{t}$ and the other $Install$ atoms false. This is not a cheapest solution (lowest cost). By calling optimize using cost-term $Cost$, the DualBoot is dropped.

Subtask 7: Explanation It is clear that a whole variety of options can be developed to provide different kinds of explanations to a user. If a user supplies an inconsistent value for a parameter, options can range from calculating a minimal inconsistent subset of the theory T as in [18,20,25], to giving a proof of inconsistency as in [15], to calculating a minimally precise partial configuration that has this inconsistency. We look at a derived logical inference for this last option.

Definition 5. ***Calculating minimal inconsistent structures.***
UnsatStructure*$(T, \mathcal{S})$ is a derived inference with as input a theory T and*

a partial structure $\mathcal{S}$ that cannot be extended to a model of T and as output all minimal (partial) structures $\mathcal{S}' \leq_p \mathcal{S}$ such that $\mathcal{S}'$ cannot be extended to a model I of T. Formally[2]*, we return:*

$$min_{\leq_p}\{\mathcal{S}' | \mathcal{S}' \leq_p \mathcal{S} \wedge \neg(\exists I \geq_p \mathcal{S}' \wedge I \vDash T)\}$$

Complexity of deciding if a set is the set of minimal inconsistent structures is Δ_2^p.

Example 7. Say a secretary wants to install all software. This is not possible, so he asks for an explanation. Running $UnsatStructure$ on the theory of Example 2 and that structure extended with $Requester = Secretary$ and $Install(software)$ true for all software will return a set with among others a structure in which $Requester = Secretary$, $Install(Windows)$ and $Install(Linux)$ are true, since a secretary does not have the budget to install two operating systems.

Subtask 8: Backtracking If a value for a parameter is not consistent, the user has to choose a new value for this parameter, or backtrack to revise a value for another parameter. In Section 3.2 we discussed three options of increasing complexity for implementing backtracking functionality. Erasing a value for a parameter is easy to provide in our KB system, and since this is a generalization of a back button (erasing the last value) we have a formalization of the first two options. Erasing a value d for parameter t in a partial structure $\mathcal{S}$ is simply modifying $\mathcal{S}$ such that $(t = d)^{\mathcal{S}} = \mathbf{u}$. As with explanation, a number of more complex options can be developed. We look at one possibility. Given a partial configuration $\mathcal{S}$, a parameter p and a value d that is inconsistent for that parameter, calculate a minimal set of previous choices that need to be undone such that this value is possible for this parameter. We can use Definition 5 and calculate $UnsatStructure(T \wedge (t = d), \mathcal{S})$. This inference calculates a set of minimal sets of previous choices that need to be undone. Backtracking over one of the sets in this set results in a maximal partial subconfiguration $\mathcal{S}'$ of $\mathcal{S}$ such that d is a possible value for t in $\mathcal{S}'$.

5 Proof of Concept

5.1 Implementation

Overview During the configuration process, the user specifies his choices step-by-step. As argued in the introduction, a configurator tool can support the user in many ways: displaying the cost of the current partial configuration, propagating the impact of the choices of the user, presenting remaining possible values for variables, explaining why certain choices are impossible, checking validity of

[2] We note that $\leq_p$ is a partial order and denote $min_{\leq_p}$ for all minimal elements of a set according to that order.

a finished configuration, completing the don't cares of a user (potentially optimizing a cost function), etc. This work started as a feasibility study about using a KB system for solving interactive configuration problems. In this section we will describe the developed application and implementation, based on the IDP system for the back-end, together with a GUI made in QML [16] as front-end.[3] This was done in cooperation with Adaptive Planet, a consulting company [1] which developed the user interface and an international banking company, who provided us with a substantial configuration problem to test our implementation. The goal was to develop a highly customizable application, for general configuration problems. The GUI is a blank canvas, which is unaware of the configuration problem at hand. The IDP KB system has a knowledge base (a theory), containing all domain knowledge, and a set with all parameters (uninterpreted terms) and IDP is used for all the inferences on that knowledge base, which provide the functionalities of the subtasks discussed in Section 4. The developed application had 300 parameters and 650 constraints. This domain knowledge was distilled from a spreadsheet that the banking company currently uses for their interactive configuration tasks. Two user interfaces are available for the user to choose from:

Wizard In the wizard interface, the user is interrogated and he answers on subsequent questions selected by the system, using the $GetOpenTerms$ inference. An important side note here is that the user can choose not to answer a specific question, for instance because he cannot decide as he is missing relevant information or because he is not interested in the actual value (at this point). These parameters can be filled in at a later timepoint by the user, or the system can fill in all parameters using autocompletion.

Drill-Down In the drill-down interface, the user sees a list of the still open parameters, and can pick which one he wants to fill in next. This interface is useful if the user is a bit more knowledgeable about the specific configuration and wants to give the values in a specific order.

In both interfaces the user is assisted in the same way when he enters data. When he or the system selects a parameter, he is provided with a dropdown list of the possible values, using the $GetConsistentValues$ inference. Before committing to a choice, he is presented with the consequences of his choice, using the calculate consequences inference. The nature of the system guarantees a correct configuration and will automatically give the user support using all information it has (from the knowledge base, or received from the user).

Evaluation When evaluating the quality of software (especially when evaluating declarative methods), scalability (data complexity) is often seen as the most important quality metric. Naturally when using an interactive configuration system, performance is important. However, in the configuration community it is

[3] More info about this implementation, a downloadable demo and another example of a configuration system developed with IDP as an engine (a simpler course configuration demo) can be found at: http://www.configuration.tk

known that reasoning about typical configuration problems is relatively easy and does not exhibit real exponential behavior [21]. In this experiment (a configuration task with 300 parameters and 650 constraints), our users reported a response time of a half second on average with outliers up to 2 seconds. Note that the provided implementation was a naive prototype and optimizing the efficiency of the implemented algorithms is still possible in a number of ways. Also, it is reasonable to expect the number of parameters to be limited, since humans need to fill in the configuration in the end. When developing a configuration system, challenges lie in the complexity of the knowledge, its high volatility and the complex functionalities to be built. In such cases, more relevant than scalability are the standard metrics of software engineering: providing good functionality, maintainability, reuse and extensibility.

Maintainability and reuse. The information used in an IC system is volatile, it is for example depending on ever-changing company policies. As such, it is vital that when that information changes, the system can be easily adapted. When using custom software, all tasks using domain knowledge (like rules and policies) need their own program code. The domain knowledge is scattered all over the program. If this policy changes, a programmer has to find all snippets of program code that are relevant for guarding this policy and modify them. This results in a system that is hard to maintain, hard to adapt and error-prone. Every time the domain knowledge changes, a whole development cycle has to be run through again. The development of a KB system with a centrally maintained knowledge base makes the knowledge directly available, readable and adaptable.

Extensibility. Supporting all subtasks expressed above is important for a good configuration system, but it is also important to have the possibility to accommodate new subtasks. A good system should be easily extensible with new functionalities, preferably without duplicating domain knowledge. This is one of the key points of a KB system. New inferences can be developed and added, independent from the domain knowledge.

Functionality. For evaluating functionality, industrial partners involved in this project have tested the proof of concept and compared with their conventional software solutions. The most common approach to developing configuration tools is building custom software. Other frequently used technology to handle interactive configuration problems are spreadsheets and business rules systems. When starting this project, the users had the following major issues with these systems, for which conceptual, general solutions were given by our approach:

– **Unidirectional dataflow:** All these systems have an obligatory unidirectional dataflow. This fixes beforehand which parameters are input and which parameters are output. However, given a problem statement, it is not natural to make a distinction between input and output. Different users may have different information or different needs and regard different parameters as input. In our approach, this distinction is not made at all by our inferences.

– **Incomplete knowledge:** These systems have problems reasoning with incomplete knowledge, i.e., rules and functions can only compute their result when their input is complete and they also cannot use partial knowledge to deduce (partial) new knowledge, e.g., to eliminate configuration options. Our language does by nature accommodate for partial knowledge, and is able to represent every intermediate partial configuration. These partial configurations are used by the inferences to calculate possible total configurations, consequences, etc.

6 Related Work

In different branches of AI research, people have been focusing on configuration software in different settings. Axling et al. [3] represent domain knowledge in the SICStus Object Language and have a configuration system specific for configuring physical objects, e.g., computers. An ontology based method was also proposed in by Vanden Bossche et al. [22] using OWL. The first reason these approaches are less general is that it is precisely the goal of the KB paradigm to reuse the knowledge for different reasoning tasks. All these approaches are focused towards one specific inference: ontologies are focused on deduction, Prolog and rule systems are focused on backward/forward chaining, etc.

Tiihonen et al. developed a configuration system WeCoTin [21] WeCoTin uses Smodels, an ASP system, as inference engine, for propagating consequences of choices. In 2004, Hadzic et al. [9] started working on solving different aspects of interactive configuration. They described solutions for these problem using knowledge compilation techniques such as binary decision diagrams (BDD) and using Boolean satisfiability solving (SAT). Hadzic et al. [9] stressed the importance of a distinction between *configuration knowledge* and the *configuration task*. This is similar to our separation of concerns by separating knowledge from computation. The authors also implemented solvers and systems for solving interactive configuration and interactive reconfiguration in later work [10]. Overall, the goal of their work is to develop different algorithms to solve different aspects of configuration problems and not to study an abstract reasoning framework in which knowledge and computation are separated. The contributions of this paper are different: we analyzed IC problems from a Knowledge Representation point of view. It is a discussion of possible approaches and the importance of this point of view. We made a study of desired functionalities for an IC system and how we can define logical reasoning tasks to supply these functionalities. In this project a more expressive language was used than in other work that we are aware of. Subbarayan et al. [19] for example use propositional logic, that is extended to CP by Andersen et al. [2]. The expressivity of the language is crucial for the usability of the approach. It allows us to address a broader range of applications, moreover it is easier to formalize and maintain the domain knowledge. A first approach in using the KB paradigm for IC, was done by Vlaeminck et al. [23], also using the FO($\cdot$) IDP project. Our work extends this, by analyzing a real-life application and discussing new functionalities.

7 Challenges and Future Work

Interactive configuration problems are part of a broader kind of problems, namely service provisioning problems. Service provisioning is the problem domain of coupling service providers with end users, starting from the request until the delivery of the service. Traditionally, such problems start with designing a configuration system that allows users to communicate their wishes, for which we provided a knowledge-based solution. After all the information is gathered from a user, it is still necessary to make a plan for the production and delivery of the selected configuration. Hence the configuration problem is followed by a planning problem that shares domain knowledge with the configuration problem but that also has its own domain knowledge about providers of components, production processes, etc. This planning problem then leads to a monitoring problem. Authorisations could be required, payments need to be checked, or it could be that the configuration becomes invalid mid-process. In this case the configuration needs to be redone, but preferably without losing much of the work that is already done. Companies need software that can manage and monitor the whole chain, from initial configuration to final delivery and this without duplication of domain knowledge. This is a problem area where the KB approach holds great promise but where further research is needed to integrate the KB system with the environment that the company uses to follow up its processes.

Other future work may include language extensions to better support configuration-like tasks. A prime example of this are templates [5]. Oftentimes the theory of a configuration problem contains lots of constraints which are similar in structure. It seems natural to introduce a language construct to abstract away the common parts. Another useful language extension is reification, to talk about the symbols in a specification rather than about their interpretation. Reification allows the system to reason on a meta level about the symbol and for example assign symbols to a category like "Technical" or "Administrative".

8 Conclusion

The KB paradigm, in which a strict separation between knowledge and problem solving is proposed, was analyzed in a class of knowledge intensive problems: interactive configuration problems. As we discussed why solutions for this class are hard to develop, we proposed a novel approach to the configuration problem based on an existing KB system. We analyzed the functional requirements of an IC system and investigated how we can provide these, using logical inferences on a knowledge base. We identified interesting new inference methods and applied them to the interactive configuration domain. We studied this approach in context of a large application, for which we built a proof of concept, using the KB system as an engine, which we extended with the new inferences. As proof of concept, we solved a configuration problem for a large banking company. Results are convincing and open perspectives for further research in service provisioning.

References

1. Adaptive planet. `http://www.adaptiveplanet.com/`.
2. Henrik Reif Andersen, Tarik Hadzic, and David Pisinger. Interactive cost configuration over decision diagrams. *J. Artif. Intell. Res. (JAIR)*, 37:99–139, 2010.
3. Tomas Axling and Seif Haridi. A tool for developing interactive configuration applications. *Journal of Logic Programming*, 19, 1994.
4. Bart Bogaerts, Joachim Jansen, Maurice Bruynooghe, Broes De Cat, Joost Vennekens, and Marc Denecker. Simulating dynamic systems using linear time calculus theories. *TPLP*, 14(4-5):477–492, 7 2014.
5. Ingmar Dasseville, M. vd Hallen, G. Janssens, and M. Denecker. Semantics of templates in a compositional framework for building logics. abs/1507.06778, 2015.
6. Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014.
7. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, April 2008.
8. Marc Denecker and Joost Vennekens. Building a knowledge base system for an integration of logic programming and classical logic. In María García de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 71–76. Springer, 2008.
9. Tarik Hadzic. A BDD-based approach to interactive configuration. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*, page 797. Springer, 2004.
10. Tarik Hadzic and Henrik Reif Andersen. Interactive reconfiguration in power supply restoration. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 767–771. Springer, 2005.
11. Neil Immerman and Moshe Y. Vardi. Model checking and transitive-closure logic. CAV '97, pages 291–302, London, UK, UK, 1997. Springer-Verlag.
12. Stephen Cole Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
13. David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.
14. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007.
15. Enrico Pontelli and Tran Cao Son. *Justifications* for logic programs under answer set semantics. In Sandro Etalle and Mirosław Truszczyński, editors, *ICLP*, volume 4079 of *LNCS*, pages 196–210. Springer, 2006.
16. Qml. `http://qmlbook.org/`.
17. D. Schneeweiss and P. Hofstedt. Fdconfig: A constraint-based interactive product configurator. In *Appl of Decl. Programming and Knowledge Management*. 2013.
18. Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*, pages 94–105. IEEE Computer Society, 2003.
19. S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, H. Hulgaard, and J. Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proc. of Workshop on CSP Techniques with Immediate Application, CP04*, 2004.

20. Tommi Syrjänen. Debugging inconsistent answer set programs. In Jürgen Dix and Anthony Hunter, editors, *NMR*, pages 77–84, 2006.
21. Juha Tiihonen, Mikko Heiskala, Andreas Anderson, and Timo Soininen. Wecotin - A practical logic-based sales configurator. *AI Commun.*, 26(1):99–131, 2013.
22. Michel Vanden Bossche, Peter Ross, Ian MacLarty, Bert Van Nuffelen, and Nikolay Pelov. Ontology driven software engineering for real life applications. In *3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2007.
23. Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 141–148. ACM, 2009.
24. Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: A model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165. ACCO, 2008.
25. Johan Wittocx, Hanne Vlaeminck, and Marc Denecker. Debugging for model expansion. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 296–311. Springer, 2009.